CHAPTER 2: TYPES, OPERATORS AND EXPRESSIONS

Variables and constants are the basic data objects manipulated in a program. Declarations state what the variables are, what type they have, and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. These are the topics of this chapter.

It's hard to present these basic objects entirely in the context of complete examples, so there will be fewer programs and more snippets and fragments and rules than we like. This can make dry reading, but bear with us; significant examples will start again in Chapter 3.

2.1 Variable Names

Although we didn't come right out and say so, there are some restrictions on variable names. Names are made up of letters and digits; the first must be a letter. The underscore '_' counts as a letter. Upper and lower case are different; traditional C practice is to use lower case for variable names, and upper case for symbolic constants.

Only the first eight characters of a name are significant; for external names such as function names, only seven are significant. (These numbers may vary from machine to machine.) Furthermore, words like `if`, `else`, `int`, `float`, etc., are *reserved;* that is, you can't use them as variable names. (They must be in lower case.)

Naturally it's wise to choose variable names that mean something, that are related to the function of the variable, and that are unlikely to get mixed up typographically. Anyone who uses `l1` and `ll` in the same program deserves what will inevitably happen.

2.2 Data Types and Sizes

There are only a few basic data types in C.

`char`	a single byte, capable of holding one character in the local character set.
`int`	an integer, typically reflecting the natural size of integers on the host machine.

float	single precision floating point.
double	double precision floating point.

In addition, there are a number of qualifiers which can be applied to int's: short, long, and unsigned. short and long refer to different sizes of integers; unsigned implies that the number is to be treated as a logical quantity, not an arithmetic one. The declarations for the qualifiers look like

```
short int x;
long int y;
unsigned int z;
```

The word int can be omitted in such situations, and typically is.

The precision of these objects depends on the machine at hand; the table below shows some representative values.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8-32
char	8 bits	9 bits	8 bits	8 bits
	ascii	ascii	ebcdic	ascii
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

The intent is that short and long should provide different lengths of integers where practical; int will normally reflect the most "natural" size for a particular machine. As you can see, each compiler is free to interpret short and long as appropriate for its own hardware. About all you should count on is that short is no longer than long.

2.3 Constants

Constants of these various types can be specified. int and float constants have already been disposed of, except to note that the usual

```
123.456e-7
```

or

```
0.12E3
```

scientific notation for float's is also legal. Every floating point constant is taken to be double, so the "e" notation serves for both float and double.

Long constants are written in the style 123L. A lower case l can also be used, but it's hard to distinguish from the digit 1. An ordinary integer constant that is too long to fit in an int is also taken to be a long.

There is a notation for octal and hexadecimal constants: a leading 0 (zero) or 0x on an int or long constant implies octal or hex respectively. For example, decimal 31 can be written as 037 in octal and 0x1f in hex.

A *character constant* is a single character written within single quotes, as in 'x'. The internal value of a character constant is the numeric value of the character in the machine's character set. These values can participate in numeric operations just as any other numbers, although they are most often used in strings and in comparisons with other characters. A later section talks about conversion rules.

Certain non-graphic characters, like newline, tab, etc., can be represented in character constants by escape sequences like \n, \t, \0, \\ (backslash), \' (single quote), etc., which look like two characters, but are actually only one.

A *constant expression* is an arithmetic expression that involves only constants. Such expressions can be evaluated at compile time, rather than run time, and accordingly may be used in any place that a constant may be. Examples are

```
char  line[MAXLINE+1];

      twopi = 2 * 3.141592654;
```

A *string constant* is a sequence of zero or more characters surrounded by double quotes, as in

```
"I am a string"
```

or

```
""      /* a null string */
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used for character constants apply in strings; of course \" represents the double quote.

Technically, a string is an array whose elements are single characters. The compiler automatically places the null character \0 at the end of each such string, so programs can conveniently find its end. This representation means that there is no real limit to how long a string can be, but programs have to scan one completely to determine its length. The physical storage requied is thus one more than the number of characters written between the quotes. The function strlen computes the length of a character string, excluding the terminal \0.

r

```
strlen(s)      /* return length of s */
char s[ ];
{
        int i;

        i = 0;
        while (s[i] != '\0')
                i++;
        return(i);
}
```

Be careful to distinguish between a character constant and a string that contains a single character: "\n" is not the same as '\n'.

2.4 Declarations

All variables must be declared before use. A declaration names the type of variable, and is followed by a list of one or more variables of that type, as in

```
int lower, upper, step;
char  c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could equally well be written as

```
int     lower;
int     upper;
int     step;
char    c;
char    line[1000];
```

This latter form takes more room, but is convenient for adding a comment to each declaration or for subsequent modifications.

Variables may also be initialized in their declaration, although there are some restrictions. If the name is followed by an equals sign '=' and a constant, that serves as an initializer, as in

```
int     i = 0;
float   twopi = 2 * 3.141592654;
char    quote = '\'';
```

Some early versions of the compiler allow the '=' to be omitted; some only allow external variables to be initialized.

If the variable in question is external or static, the initialization is done once only, conceptually before the program starts executing. Local variables are initialized each time the function they are in is called. Unitialized variables have undefined values, usually garbage.

Whether to initialize a variable in its declaration or in the code which follows is not always clear-cut. Consider

```
int     i = 0;

while (s[i] != '\0')
        i++;
```

and

```
int     i;

i = 0;
while (s[i] != '\0')
        i++;
```

We have a mild preference for the second version because the initialization of i is closer to the code that manipulates it.

We will discuss initialization more as new objects are introduced.

2.5 Arithmetic Operators

The arithmetic operators are +, −, *, /, and the modulus operator %. Integer division truncates. The expression

```
a % b
```

produces the remainder when a is divided by b, and thus is zero when b divides a exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 *are* leap years. Therefore

```
if ((year%400 == 0) || (year%100 != 0 && year%4 == 0))
        it's a leap year
else
        it's not
```

% cannot be applied to float or double. For both / and %, if either operand is negative, the results are machine dependent.

The + and − operators have the same precedence, which is lower than the (identical) precedence of *, /, and %. Arithmetic operators group left to right. The order of evaluation is not specified for associative and commutative operators like * and +; the compiler is free to rearrange even parenthesized computations as it wishes. Thus a+(b+c) may well be evaluated as (a+b)+c.

Any underflow or overflow that occurs as an expression is being evaluated is silently ignored.

2.6 Relational Operators

The relational operators are, in order of decreasing precedence,

```
>  >=  <  <=
==  !=
```

Operators on the same line have the same precedence. Relationals have lower precedence than arithmetic operators, so expressions like i < lim−1 are taken as i < (lim−1), as would be expected.

More interesting are the logical operators && and ||. Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. These properties are both critical to writing programs that work. For example, here is a loop from the input function getline which we wrote in Chapter 1.

```
for (i = 0; i < lim−1 && (c = getchar()) != '\n' && c != EOF; i++)
        s[i] = c;
```

Clearly, it is necessary to check *first* that there is room in the array s to store any new character, so the test i < lim−1 must be made first. Not only that, but if this test fails, we must not go on and read another character.

Similarly, it would be unfortunate if c were tested against EOF before getchar was called: the call *must* be made before the character in c is tested.

The precedence of && is greater than that of ||, and both are lower than the relationals, so that expressions like

```
i < lim−1 && (c = getchar()) != '\n' && c != EOF
```

need no extra parentheses. But notice again that the precedence of != is higher than assignment, so parentheses are needed around

```
(c = getchar()) != '\n'
```

Exercise 2-1: Write the for loop above without &&. □

2.7 Type Conversions

What happens if you say

```
int i;

i = 'x' + 3e5
```

Not that anyone would, perhaps, but the question of type conversions requires some discussion.

First, char's and int's may be freely intermixed in arithmetic expressions: every char in an expression is automatically converted to an int. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by the routine atoi which converts a string of ASCII digits into its numeric equivalent.

```
atoi(s)/* convert s to integer */
char s[ ];
{
        int i, n;

        n = 0;
        for (i = 0; s[i] >= '0' && s[i] <= '9'; i++)
                n = 10 * n + s[i] - '0';
        return(n);
}
```

The expression

```
s[i] - '0'
```

gives the numeric value of the character stored in s[i] only if the digits are contiguous and in increasing order as characters; fortunately this is true for all known character sets.

Exercise 2-2: Are the assignments

```
    n *= 10 + s[i] - '0'
```

haven't introduced *= yet

and

```
    n = 10 * n + s[i] - '0'
```

equivalent? □

Another example is the routine lower which converts a single character to lower case *for the ASCII character set only.*

```
lower(c)
char c;
{
        if (c >= 'A' && c <= 'Z')
                return(c - 'A' + 'a')
        else
                return(c);
}
```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and are contiguous — there is nothing but letters between 'a' and 'z'. This latter observation is *not* true of the EBCDIC alphabet (IBM), so this code fails on such systems — it converts more than letters.

Exercise 2-3: Write the corresponding function upper, which converts lower case letters to upper case. □

There is one subtle point about the conversion of characters to integers. If a character is negative does it become a negative integer ("sign extension"), or is it positive? Regrettably, this varies from machine to machine. It is true that by definition, any character in the machine's standard character set will never be negative, so the problem won't arise there. But for arbitrary bit patterns stored in character variables, it can.

Another useful form of automatic type conversion is that relational expressions like i > j and logical expressions connected by && and | are defined to have value 1 if true, and 0 if false. Thus the assignment

```
leap = (year%400 == 0) || (year%100 != 0 && year%4 == 0)
```

has the value 1 for leap years and 0 for non-leap years.

Implicit arithmetic conversions work much as expected. We have already seen char's become int's, and int's become float's, as in

```
cent = 5.0/9.0 * (fahr - 32)
```

All floating point arithmetic in C is done in double precision, so all float's in an expression are converted to double.

In general, if an operator like + or * which takes two operands (a "binary operator") has operands of different types, the "lower" type is *promoted* to the "higher" type before the operation proceeds. The result is of the higher type. More formally, for all arithmetic operators, the following sequence of conversion rules is applied.

> char is converted to int and float is converted to double.
> Then if either operand is double, the other is converted to double, and the result is double.
> Then if either operand is long, the other is converted to long, and the result is long.
> Then if either operand is unsigned, the other is converted to unsigned, and the result is unsigned.
> Otherwise the operands must be int, and the result is int.

Conversions also take place across assignment statements; the value of the right side is converted to the type of the left. If x is float and i is int, then

```
x = i;
```

and

```
i = x;
```

both cause conversions; float to int causes truncation of any fractional part.

Finally, explicit type conversions can be put in any expression with a construct called a *cast*. In the construction

(*type*) expression

the expression is converted to the named type. For example, the library routine sqrt expects a double argument, and will produce nonsense if inadvertently handed something else. So if n is an integer,

```
sqrt((double) n)
```

converts n to double before passing it to sqrt.

Early versions of C do not provide the cast operation; in that case, you must do your own conversions by assigning to explicit temporary variables of the proper type.

2.8 Increment and Decrement Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand; the decrement operator -- subtracts 1. We have frequently used ++ to increment variables, as in

```
    if (c == '\n')
        nl++;
```

The unusual aspect is that ++ and -- may be either prefix (before the variable, as in ++i), or postfix (after the variable: i++). In both cases, the effect is to increment i, but the *value* of the expression ++i is i *after* it is incremented, while the value of i++ is i *before* it is incremented. Thus if i is 5, then

```
x = i++
```

sets x to 5, but

```
x = i++   ++i
```

sets x to 6. In both cases, i becomes 6.

In a context where no value is wanted, just the incrementing effect, as in

```
    if (c == '\n')
        nl++;
```

choose prefix or postfix according to taste. But there are situations where one or the other is specifically called for. For instance, consider the function squeeze(s, c) which removes all occurrences of the character c from the string s.

```
squeeze(s, c)       /* delete all c from s */
char s[ ], c;
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Each time a non-c occurs, it is copied into the current j position, and only then is j incremented to be ready for the next character.

Of course, we could have written the longer form

```
        if (s[i] != c) {
              s[j] = s[i];
              j++;
        }
```

but the shorter form is better, and, after a bit of experience, just as easy to read.

As another example, perhaps more compelling, here is a function called **strcat(s, t)**, which concatenates the string **t** to the end of the string **s**. **strcat** believes that there is enough space in **s** to hold the combination.

```
strcat(s, t)    /* concatenate t to end of s */
char s[ ], t[ ];
{
      int i, j;

      i = j = 0;
      while (s[i])   /* find end of s */
            i++;
      while (s[i++] = t[j++])  /* copy t */
            ;
}
```

The postfix ++ is applied to both i and j to make sure that they are both in position for the next pass through the loop.

The test could be written as

```
                           i++   j++
      while (s[i] = t[i]) != '\0')  /* copy t */
```

but since \0 is zero, and since **while** tests whether the parenthesized expression is zero or not, the != '\0' is redundant and can be omitted. Although you might argue that this is bad form, the \0 is very often elided.

2.9 Bitwise Logical Operators

C provides a full set of bitwise logical operators in addition to the usual arithmetic ones. These include & (bitwise AND), | (bitwise inclusive OR), and ^ (bitwise exclusive OR). The classic application of & is masking off everything but the last 7 bits, to make an integer into an ASCII character:

```
c = n & 0177;
```

You should carefully distinguish the bitwise operators & and | from && and ||, which imply left-to-right evaluation of a truth value. For example, if x is 1 and y is 2, then x & y is zero while x && y is one.

The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand. Thus x << 2 shifts x left by two positions, equivalent to multiplication by 4. Whether a shift is logical or arithmetic depends: if the variable in question is **unsigned**, then the shift is logical; otherwise it is arithmetic on some machines, so beware.

The unary negation operator ! converts a non-zero or true operand into 0, and a zero or false operand into 1. A common use of ! is in constructions like

```
if (!inword)
```

rather than

```
if (inword == 0)
```

It's hard to generalize about which form is better. Constructions like !inword read quite nicely ("if not in word"), but more complicated ones can be hard to understand.

The unary operator ~ gives the one's complement of integers; that is, it converts each 1 bit into a 0 bit and vice versa. This operator typically finds use in expressions like

```
x &= ~077
```

which masks the last six bits of x to zero.

2.10 Assignment Operators

Expressions like

```
i = i + 10
```

abound in typical programs. Expressions like these, where the left hand side is repeated on the right, can be written in the compressed form

```
i += 10
```

using an *assignment operator* like +=.

Any binary operator op, that is, one that has a left and right operand, has a corresponding assignment operator op=. op is one of

```
+  -  *  /  %  <<  >>  &  ^  |
```

If e1 and e2 are expressions, then

```
e1 op= e2
```

is equivalent to

```
e1 = e1 op (e2)
```

except that e1 is computed only once. The parentheses around e2 should be noted — for example,

```
x *= y + 1
```

is actually

```
x = x * (y + 1)
```

rather than

```
x = x * y + 1
```

The type of an assignment expression like `i += 10` is the type of its left operand.

The conciseness of an assignment operator is not much saving for an expression as simple as `i += 10`, but for something like

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

it is well worthwhile. It makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same. And it may help the compiler to produce more efficient code.

Exercise 2-4: Rewrite the **log2** function of Chapter 1 with an assignment operator. □

As an example that combines logical and assignment operators, here is a function called **bitcount** which counts the number of 1 bits in its integer argument.

```
bitcount(n)  /* count 1 bits in n */
unsigned n;
{
        int b;

        for (b = 0; n != 0; n >>= 1)
                if (n & 01)
                        b++;
        return(b);
}
```

Declaring the argument to be **unsigned** ensures that when **n** is right-shifted, vacated bits will be filled with zeros, not with sign bits; regardless of the machine the program is run on.

The loop can also be written as

```
        for (b = 0; n != 0; n >>= 1)
                b += n % 2;
```

since `n % 2` is 1 if **n** is odd, that is, if its bottom bit is turned on, and zero otherwise.

Early versions of C used the form **=op** instead of **op=** for assignment operators. This leads to some nasty ambiguities, typified by

```
x=-1
```

Does this decrement **x** or set it to -1? The only way to be sure is to add a space at the right place, to make either

```
x =- 1
```

or

```
x = -1
```

If your C compiler supports the new form, use it.

Exercise 2-5: AND'ing a value n with n−1 deletes the rightmost 1 bit in n. Use this fact to write a faster version of bitcount. □

2.11 Assignments as Values

We have already used the fact that the assignment statement has a value and can occur in expressions; the most common example is

```
while ((c = getchar( )) != EOF)
        ...
```

The other assignment operators (+=, −=, etc.) can also be used in this way, although it is a less frequent occurrence. As an illustration, here is a line from a program that we will discuss in detail in Chapter 4:

```
val += (c - '0') / (den *= 10.0);
```

den is multiplied by 10 before it in turn is used to divide c − '0'; the result is then added to val.

Increment and decrement operators and nested assignment statements cause "side effects" — some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which parts of the expression are evaluated. One unhappy situation is typified by

```
a[i++] = i;
```

The question is whether i is incremented before or after it's assigned to a. The compiler can obviously do this in different ways, and generate different answers depending on its interpretation. Normally this kind of code is only written inadvertently, but you should be aware of the possibility.

Exercise 2-6: Write a program to make *all* ASCII non-printing characters visible. □